CHAPTER 1. INTRODUCTION TO THE ALGORITHMICS

1. Definition of algorithm

An algorithm is an abstract and ordered description of all the actions that must be performed, as well as the description of the data that must be manipulated by those actions, in order to arrive at the solution of a problem. It should:

- Be independent of both the programming language in which it is expressed, and the computer on which it will be executed.
- Be clear and simple.
- Indicate the order of completion of each step.
- Have a finite number of steps (as well as a beginning and an end).
- Be flexible (to facilitate maintenance).
- Be defined (so if you follow an algorithm N times with the same input data, you should get the same result N times).

Characteristics:

- Algorithms solve problems.
- A problem usually has many instances (sometimes infinite).
- Algorithms must work correctly in all cases of the problem they claim to solve.
- A single erroneous case causes the algorithm to be incorrect.
- We usually find more than one way to solve a problem, that is, there are several algorithms that solve the same problem.

Example: multiplication of two numbers:

- In Spain:

```
    981
   1234
   ----
   3924
  2943
 1962
 981
-------
1210554
```

- In England:

```
    981
   1234
-------
 981
 1962
  2943
   3924
-------
1210554
```

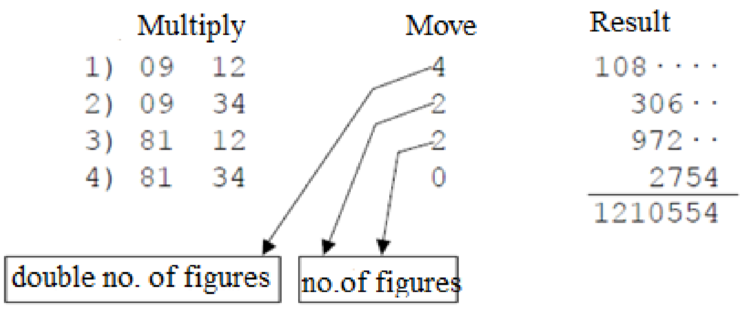
Both methods are very similar and we can call them the "classical" multiplication algorithm.

Multiplication “in the Russian way”:

✓	981	1234	1234
	490	2468	
✓	245	4936	4936
	122	9872	
✓	61	19744	19744
	30	39488	
✓	15	78976	78976
✓	7	157952	157952
✓	3	315904	315904
✓	1	631808	631808
			1210554

It has the advantages that you do not have to store partial products and that you only have to multiply and divide by two.

Another algorithm: in this case both numbers must have the same number of figures and this must be power of 2. The first step is to divide both numbers in half and make four products:



	Multiply		Move	Result
1)	09	12	4	108····
2)	09	34	2	306··
3)	81	12	2	972··
4)	81	34	0	2754
				1210554

That is, a product of four-digit numbers is reduced in four products of two-digit numbers, several displacements and a sum. The products of two-digit numbers can be made in the same way:

	Multiply		Move	Result
1)	0	1	2	0··
2)	0	2	1	0·
3)	9	1	1	9·
4)	9	2	0	18
				108

It is an example of the "divide and conquer" technique.

As the different multiplication algorithms have been presented, the classical algorithm does not improve in efficiency. But, it can be improved: it is possible to reduce a product of two numbers of many figures to 3 (instead of 4) products of numbers of half the number of figures, and this one does improve the classical algorithm. And even faster methods are known to multiply very large numbers. We need some way to measure the efficiency of an algorithm in order to find "the best" algorithm. The task of finding "the best" algorithm is the basis of what is known as algorithmics whose main objectives are:

- Perform a systematic treatment of fundamental techniques for the design and analysis of efficient algorithms.
- Study the properties of the algorithms and choose the most appropriate solution in each situation. This saves time and money. In many cases, a good choice makes the difference between being able to solve a problem and not being able to do so.

2. Efficiency of the algorithms

The unit to measure the efficiency of the algorithms is based on the Principle of Invariance, which says that "two different implementations of the same algorithm will not differ in efficiency more than, at most, in a multiplicative constant". This means that if two implementations consume t1(n) and t2(n) units of time, respectively, when a case of size n is solved, then there is always a positive constant c such that t1(n)<=ct2(n) , provided that n is large enough. The Principle of Invariance is valid, regardless of the computer used, the programming language used and the ability of the programmer (assuming he does not modify the algorithm).

The fact that the runtime depends on the input tells us that this time must be defined based on that input (or the size of the input). Therefore, T(n) is the execution time of a program for an input of size n, and also for that of the algorithm on which it is based.

We say that an algorithm consumes a time of order t(n), for a given function t, if there is a positive constant c and an implementation of the algorithm capable of solving any case of the problem in a time bounded superiorly by ct(n) units of time, where n is the size (or the value, for numerical problems) of the case considered.

Asymptotic notation

An algorithm with a runtime T(n) is said to be of order O(f(n)) if there exists a positive constant c and an integer n_0 such that for all n>=n_0 then T(n)<=cf(n)

For example, if T(0)=1, T(1)=4 and T(n)=$(n+1)^2$. Then T(n) is O(n^2), then if we take n_0=1 and c=4, it is verified that for all n>=1 then $(n+1)^2$<=$4n^2$

If an algorithm has an execution time O(f(n)), f(n) will be called the Growth Rate. O(f(n)) (which reads "the order of f(n)") is the set of all functions t(n) bounded superiorly by a positive real multiple of f(n), given n large enough (greater than some threshold n_0) When T(n) is O(f(n)), we are giving a higher bound for the execution time, which we will always refer to the worst case.

Rule of the addition

Suppose, first of all, that T1(n) and T2(n) are the execution times of two program segments, P1 and P2, that T1(n) is O(f(n)) and T2(n) is O(g(n)). Then the execution time of P1 followed by P2, ie T1(n)+T2 (n), is O(max(f(n), g(n))).

For example, we have an algorithm consisting of 3 stages, in which each of them can be an arbitrary fragment of algorithm with loops and branches. Let us suppose their respective times O(n^2), O(n^3) and O(nlog n). So

- The execution time of the first two sequentially executed stages is O(max(n^2, n^3)), that is, O(n^3).

- The execution time of the three parts together is O(max(n^2, n^3, nlogn)), that is, O(n^3).

Rule of the product

If T1(n) and T2(n) are the execution times of two program segments, P1 and P2, T1(n) is O(f(n)) and T2(n) is O(g(n)) , then T1(n)T2(n) is O(f(n)g(n)). From this rule it follows that O(cf(n)) is the same as O(f(n)) if c is a positive constant, so for example O(n^2/2) is the same as O(n^2).

3. Theory of the calculation of the efficiency

Elementary operations

The measurement of the time an algorithm will be performed based on the number of elementary operations (EO) it performs. An elementary operation is one whose consumption of execution time can be bounded by a constant, that is, it does not depend on the size of the instance that is being executed. For simplicity we consider the cost of an elementary operation, or of a set of elementary operations as cost 1, that is, EO is O(1).

The following operations if applied to basic type data are elementary operations:

- Basic arithmetic operations: +, -, *, /
- Logical operations: AND, OR, NOT.
- Order operations: <,>, =
- Reading or writing
- Assignment of values
- The Return instruction

Conditionals

They are instructions as

If Cond then InstrT else InstrF eif

The cost of all the instruction is:

Cost(Cond)+Max{Cost(InstrT)+Cost(InstrF)}

Loops

For the non determined loops as

Repeat InstrR until Cond eRepeat

The cost is obtained in the following way:

Cost(InstrR)+Cost(Cond)+$\sum_{cond=False}\left(\text{Cost(InstrR)} + \text{Cost(Cond)}\right)$

For the determined loops as

For Var=Vini to Vfin Do InstrF fFor

The cost is obtained in the following way:

Max{Cost(Vini)+Cost(Vfin)}+$\sum_{Var=Vini}^{Vfin} \text{Cost(InstrF)}$

Recursive efficiency. Characteristic equation

To find the efficiency of a recursive method, we will use the method of the characteristic equation, which consists on finding the roots of the mathematical equation that represents the use of resources of a recursive method.

Examples

Determine the efficiency function and complexity order of the following codes:

a)
```
if A[1,1] = 0 Then
    for i := 1 to n do
        for j := 1 to n do
            A[i,j] := 0
else
    for i := 1 to n do
        A[i,i] := 1
```

b)
```
for i := 1 to n-1 do begin
        min := i
        for j := i+1 to n do
                if A[j] < A[min] Then
                        min := j
        temp := A[min]
        A[min] := A[i]
        A[i] := temp
end
```

c)
```
Procedure Insert (T[1..n])
for i := 2 to n do
        x:= T[i];
        j := i-1
        while j > 0 and x < T[j] do
                T[j+1] := T[j]
                j := j-1
        T[j+1] := x
End
```

d)
```
Procedure Problem3(n,v[1..n])
if (n>0) and (n<=dim) then
        for  i=1 to (n-1) do v[i]=v[i+1] end
        Problem3(n-1,v)
End
```